5. Working with Makefiles

Created: April 1, 2003
Updated: September 16, 2003

Overview

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

Building executables and libraries for a large, integrated set of software tools like the C++ Toolkit, and doing so consistently on different platforms and architectures, is a daunting task. Therefore, the Toolkit developers have expended considerable effort to design a build system based upon the *make* utility as controlled by *makefiles*. While it is of course possible to write one's own Toolkit *makefile* from scratch, it is seldom desirable. To take advantage of Toolkit experience, wisdom and alchemy to help avoid often inscrutable compilation issues:

> **we strongly advise users to work with the Toolkit's make system**

With minimal manual editing (and after invoking the ***configure*** script in your build tree) the build system adapts to your environment, compiler options, defines all relevant *makefile* macros and targers, allows for recursive builds of the entire Toolkit and targetted builds of single modules, and handles many other details that can confound manual builds.

Chapter Outline

The following is an outline of the topics presented in this chapter:

- Major Makefiles
- Makefile Hierarchy
- Meta-Makefiles
    - Makefile.in Meta Files
    - Expendable Projects

- Project Makefiles
    - List of optional packages, features and projects
- Standard Build Targets
    - Meta-Makefile Targets
    - Makefile Targets
- Makefile Macros and Makefile.mk
- Example Makefiles

Major Makefiles

Before describing the make system in detail, we list the major types of *makefiles* employed by the Toolkit:

- ***meta-makefiles*** These files exist for each project and tie together project in the Toolkit hierarchy, defining those applications and libraries a project is responsible for (possibly recursively) building.
- **Generic *makefile* Templates** (*Makefile*.in*) The ***configure*** script processes these files from the *src* hierarchy to substitute for the special tags `"@some_name@"` and make other specializations required for a given project. Note that *meta-makefiles* are typically derived from such templates.
- **Customized *makefiles*** (*Makefile.*.[lib|app]*) For each library or application, this file gives specific targets, compiler flags, and other project-specific build instructions. These files appear in the *src* hierarchy.
- **Configured *makefiles*** (*Makefile*) A *makefile* generated by ***configure*** for each project and sub-project and placed in the appropriate location in the build tree ready for use will be called a '*configured makefile*'. Note that *meta-makefiles* in the build tree may be considered 'configured'.

Makefile Hierarchy

All Toolkit *makefiles* reside in either the *src* directory as templates or customized files, or in the appropriate configured form in each of your <*builddir*> hierarchies as illustrated in Figure 1

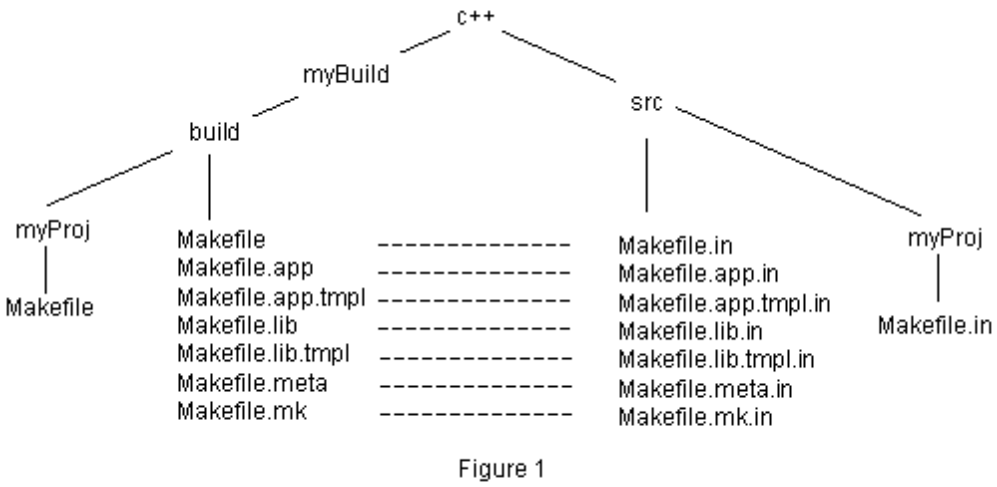

Figure 1: Makefile Hierarchy

Most of the files listed in Figure 1 are templates from the *src* directory, with each corresponding *configured makefile* at the top of the build tree. Of these, *<builddir>/Makefile* can be considered the master *makefile* in that it can recursively build the entire Toolkit. The role of each top-level *makefile* templates is summarized as follows:

- ***Makefile.in*** -- makefile to perform a recursive build in all project subdirectories
- ***Makefile.meta.in*** -- included by all makefiles that provide both local and recursive builds
- ***Makefile.mk.in*** -- included by all makefiles; sets a lot of configuration variables
- ***Makefile.lib.in*** -- included by all makefiles that perform a "standard" library build, when building only static libraries.
- ***Makefile.dll.in*** -- included by all makefiles that perform a "standard" library build, when building only shared libraries.
- ***Makefile.both.in*** -- included by all makefiles that perform a "standard" library build, when building both static and shared libraries.
- ***Makefile.lib.tmpl.in*** -- serves as a template for the project *customized makefiles* (***Makefile.*.lib[.in]***) that perform a "standard" library build
- ***Makefile.app.in*** -- included by all makefiles that perform a "standard" application build
- ***Makefile.lib.tmpl.in*** -- serves as a template for the project *customized makefiles* (***Makefile.*.app[.in]***) that perform a "standard" application build
- ***Makefile.rules.in, Makefile.rules_with_autodep.in*** -- instructions for building object files; included by most other makefiles

The project-specific portion of the *makefile* hierarchy is represented in the figure by the *meta-makefile* template *c++/src/myProj/Makefile.in*, the *customized makefilec++/src/myProj/Makefile.myProj.[app|lib]* (not shown), and the *configured makefilec++/myBuild/build/myProj/Makefile*. In fact, every project and sub-project in the Toolkit has analogous files specialized to its project; in most circumstances, every new or user project should emulate this file structure to be compatible with the make system.

Meta-Makefiles

A typical *meta-makefile* template(e.g. *Makefile.in* in your *foo/c++/src/bar_proj/* dir) looks like this:

```
# Supply Makefile.bar_u1, Makefile.bar_u2 ...
#
USR_PROJ = bar_u1 bar_u2 ...

# Supply Makefile.bar_l1.lib, Makefile.bar_l2.lib ...
#
LIB_PROJ = bar_l1 bar_l2 ...

# Supply Makefile.bar_a1.app, Makefile.bar_a2.app ...
#
APP_PROJ = bar_a1 bar_l2 ...

# Subprojects
#
SUB_PROJ = app sub_proj1 sub_proj2

srcdir = @srcdir@
include @builddir@/Makefile.meta
```

This template separately specifies instructions for user, library and application projects, along with a set of three sub-projects that can be made. The mandatory final two lines `"srcdir = @srcdir@ ; include @builddir@/Makefile.meta"` define the standard build targets.

The following topics are discussed in the subsections that follow:

- Makefile.in Meta Files
- Expendable Projects

Makefile.in Meta Files

The *Makefile.in* meta file in the project's source directory defines a kind of road map that will be used by the ***configure*** script to generate a makefile (*Makefile*) in the corresponding directory of the *build tree*. *Makefile.in* does **not** participate in the actual execution of *make*, but rather, defines what will happen at that time by directing the ***configure*** script in the creation of the *Makefile* that will be executed (see also the description of Makefile targets below).

The meta-makefile *myProj/Makefile.in* should define at least one of the following macros:

- USR_PROJ (optional) - a list of names for user-defined makefiles. This macro is provided for the usage of ordinary stand-alone makefiles which do not utilize the make commands contained in additional makefiles in the top-level *build* directory. Each *p_i* listed in `USR_PROJ = p_1 ... p_N` must have a corresponding Makefile.p_i in the project's source directory. When *make* is executed, the *make* directives contained in these files will be executed directly to build the targets as specified.

- LIB_PROJ (optional) - a list of names for library makefiles. For each library *l_i* listed in `LIB_PROJ = l_1 ... l_N`, you must have created a corresponding project makefile namedMakefile.l_i.lib in the project's source directory. When *make* is executed, these library project makefiles will be used along with *Makefile.lib* and *Makefile.lib.tmpl* (located in the top-level of the *build tree*) to build the specified libraries.

- ASN_PROJ (optional) is like LIB_PROJ, with one additional feature: Any projects listed there will be interpreted as the names of ASN.1 module specifications to be processed by datatool.

- APP_PROJ (optional) - a list of names for application makefiles. Similarly, each application (*p1, p2, ..., pN*) listed under APP_PROJ must have a corresponding project makefile named Makefile.p*.app in the project's source directory. When *make* is executed, these application project makefiles will be used along with *Makefile.app* and *Makefile.app.tmpl* to build the specified executables.

- SUB_PROJ (optional) - a list of names for subproject directories (used on recursive makes). The SUB_PROJ macro is used to recursively define *make* targets; items listed here define the subdirectories rooted in the project's source directory where *make* should also be executed.

Some additional *meta-makefile* macros (listed in Table 1) exist to specify various directory paths that *make* needs to know. The "@"-delimited tokens are substituted during configuration based on your environment and any command-line options passed to ***configure***.

Table 1. Path Specification Makefile Macros

Macro	Source	Synopsis
`top_srcdir`	`@top_srcdir@`	Path to the whole NCBI C++ package
`srcdir`	`@srcdir@`	Directory in the *source tree* that corresponds to the directory (*./*) in the *build tree* where the build is currently going on
`includedir`	`@includedir@`	Top include directory in the *source tree*
`build_root`	`@build_root@`	Path to the whole *build tree*

Macro	Source	Synopsis
builddir	@builddir@	Top build directory inside the *build tree*
incdir	@incdir@	Top include directory inside the *build tree*
libdir	@libdir@	Libraries built inside the *build tree*
bindir	@bindir@	Executables built inside the *build tree*
status_dir	@status_dir@	Configuration status files

Expendable Projects

By default, failure of any project will cause make to exit immediately. Although this behavior can save a lot of time, it is not always desirable. One way to avoid it is to run `make -k` rather than `make`, but then major problems affecting a large portion of the build will still waste a lot of time.

Consequently, the toolkit's build system supports an alternative approach:*meta-makefiles* can define *expendable* projects which should be built if possible but are allowed to fail without interrupting the build. The way to do this is to list such projects in `EXPENDABLE_*_PROJ` rather than `*_PROJ`.

Project Makefiles

When beginning a new project, the ***new_project.sh*** shell script will generate an initial *makefileMakefile.<project_name>_app* that you can modify as needed. In addition, a working sample application can also be checked out to experiment with or as an alternate template.

The ***import_projects.sh*** script is useful for working on existing Toolkit projects without needing to build the whole Toolkit. In this case things are particularly straightforward as the project will be retrieved complete with its *makefile* already configured as *Makefile.<project_name>_[app|lib]*. (Note that there is an underscore in the name, not a period as in the similarly-named *customizable makefile* from which the configured file is derived.)

If you are working outside of the source tree: In this scenario you are only linking to the Toolkit libraries and will not need to run the ***configure*** script, so a *Makefile.in* template *meta-makefile* is not required. Some of the typical edits required for the *customized makefile* are shown in the programming manual.

If you are working within the source tree or subtree: Project subdirectories that do not contain any **.in* files are ignored by the ***configure*** script. Therefore, you will now also need to create a *meta-makefile* for the newly created project before configuring your *build* directory to include the new project.

Several examples are detailed on the "Starting New Projects" page.

List of optional packages, features and projects

Table 2 displays the keywords you can list in REQUIRES in a customized application or library makefile, along with the corresponding configure options:

Table 2. Optional Packages, Features, and Projects

Keyword	Optional...	Configure option(s)
FreeTDS	FreeTDS libraries	*--without-ftds, --with-ftds=DIR*
Fast-CGI	Fast-CGI library	*--without-fastcgi*
FLTK	the Fast Light ToolKit	*--without-fltk, --with-fltk=DIR*
wxWindows	wxWindows	*--without-wxwin, --with-wxwin=DIR*
C-Toolkit	NCBI C Toolkit	*--without-ncbi-c*
SSSDB	NCBI SSS DB library	*--without-sssdb, --without-sss*
SSSUTILS	NCBI SSS UTILS library	*--without-sssutils, --without-sss*
GEO	NCBI GEO libraries	*--without-geo*
SP	SP libraries	*--without-sp*
PubMed	NCBI PubMed libraries	*--without-pubmed*
serial	ASN.1/XML serialization library and datatool	*--without-serial*
ctools	projects based on the NCBI C toolkit	*--without-ctools*
gui	projects that use the wxWindows GUI package	*--without-gui*
objects	libraries to serialize ASN.1/XML objects	*--with-objects*
app	standalone applications like ID1_FETCH	*--with-app*
internal	all internal projects	*--with-internal*
local_lbsm	IPC with locally running LBSMD	*--without-local-lbsm*

Standard Build Targets

The following topics are discussed in this section:

- Meta-Makefile Targets
- Makefile Targets

Meta-Makefile Targets

The mandatory lines from the *meta-makefile* example above,

```
srcdir = @srcdir@
include @builddir@/Makefile.meta
```

provide the build rules for the following standard *meta-makefile* targets:

- *all*:
  - run `"make -f {Makefile.*} all"` for the makefiles with the suffixes listed in macro `USR_PROJ`:

    ```
    make -f Makefile.bar_u1 all
    make -f Makefile.bar_u2 all
    ......
    ```

  - build libraries using attributes defined in the *customized makefilesMakefile.*.lib* with the suffixes listed in macro `LIB_PROJ`
  - build application(s) using attributes defined in the *customized makefilesMakefile.*.app* with the suffixes listed in macro `APP_PROJ`
- *all_r* -- first make target *all*, then run `"make all_r"` in all subdirectories enlisted in `$(SUB_PROJ)`:

  ```
  cd bar_test      && make -f Makefile all_r
  cd bar_sub_proj1 && make -f Makefile all_r
  ......
  ```

- *clean, clean_r* -- run just the same makefiles but with targets *clean* and *clean_r* (rather than *all* and *all_r*), respectively
- *purge, purge_r* -- .....with targets *purge* and *purge_r*, respectively

Makefile Targets

The standard build targets for Toolkit *makefiles* are *all, clean* and *purge*. Recall that recursive versions of these targets exist for *meta-makefiles*.

- *all* -- compile the object modules specified in the `"$(OBJ)"` macro, and use them to build the library `"$(LIB)"` or the application `"$(APP)"`; then copy the resultant [*lib*|*app*] to the [*libdir*|*bindir*] directory, respectively
- *clean* -- remove all object modules and libs/apps that have been built by *all*

- *purge* -- do *clean*, and then remove the copy of the [*libs*|*apps*] from the [*libdir*|*bindir*] directory.

The *customized makefiles* do not distinguish between recursive (*all_r, clean_r, purge_r*) and non-recursive (*all, clean, purge*) targets -- because the recursion and multiple build is entirely up to the *meta-makefiles*.

Makefile Macros and *Makefile.mk*

There is a wide assortment of configured tools, flags, third party packages and paths (see above). They can be specified for the whole build tree with the appropriate entry in *Makefile.mk*, which is silently included at the very beginning of the *customized makefiles* used to build libraries and applications.

Many *makefile* macros are supplied with defaults `ORIG_*` in *Makefile.mk*. See the list of `ORIG_%20` macros, and all others currently defined, in the *Makefile.mk.in* template for details. One should not override these defaults in normal use, but add your own flags to them as needed in the corresponding working macro; e.g., set `CXX = $(ORIG_CXX) -DFOO_BAR`.

Makefile.mk defines the following *makefile* macros obtained during the configuration process for flags (see Table 3), system and third-party packages (see Table 4) and development tools (see Table 5).

Table 3. Flags

Macro	Source	Synopsis
`CFLAGS`	`$CFLAGS`	C compiler flags
`FAST_CFLAGS`	`$FAST_CFLAGS`	(*) C compiler flags to generate faster code
`CXXFLAGS`	`$CXXFLAGS`	C++ compiler flags
`FAST_CXXFLAGS`	`$FAST_CXXFLAGS`	(*) C++ compiler flags to generate faster code
`CPPFLAGS`	`$CPPFLAGS`	C/C++ preprocessor flags
`DEPFLAGS`	`$DEPFLAGS`	Flags for file dependency lists
`LDFLAGS`	`$LDFLAGS`	Linker flags
`LIB_OR_DLL`	`@LIB_OR_DLL@`	Specify whether to build a library as static or dynamic
`STATIC`	`@STATIC@`	Library suffix to force static linkage (see example)

(*) The values of user-specified environment variables `$FAST_CFLAGS, $FAST_CXXFLAGS` will substitute the regular optimization flag *-O* (or *-O2*, etc.). For example, if in the environment: `$FAST_CXXFLAGS`=*-fast -speedy* and `$CXXFLAGS`=*-warn -O3 -std*, then in makefile: `$(FAST_CXXFLAGS)`=*-warn -fast -speedy -std*.

Table 4. System and third-party packages

Macro	Source	Synopsis
LIBS	$LIBS	Default libraries to link with
PRE_LIBS	$PRE_LIBS	??? Default libraries to link with first
THREAD_LIBS	$THREAD_LIBS	Thread library (system)
NETWORK_LIBS	$NETWORK_LIBS	Network library (system)
MATH_LIBS	$MATH_LIBS	Math library (system)
KSTAT_LIBS	$KSTAT_LIBS	KSTAT library (system)
RPCSVC_LIBS	$RPCSVC_LIBS	RPCSVC library (system)
SYBASE_INCLUDE	$SYBASE_INCLUDE	SYBASE headers
SYBASE_LIBS	$SYBASE_LIBS	SYBASE libraries
FASTCGI_INCLUDE	$FASTCGI_INCLUDE	Fast-CGI headers
FASTCGI_LIBS	$FASTCGI_LIBS	Fast-CGI libraries
NCBI_C_INCLUDE	$NCBI_C_INCLUDE	NCBI C toolkit headers
NCBI_C_LIBPATH	$NCBI_C_LIBPATH	Path to the NCBI C Toolkit libraries
NCBI_C_ncbi	$NCBI_C_ncbi	NCBI C CoreLib
NCBI_SSS_INCLUDE	$NCBI_SSS_INCLUDE	NCBI SSS headers
NCBI_SSS_LIBPATH	$NCBI_SSS_LIBPATH	Path to NCBI SSS libraries
NCBI_PM_PATH	$NCBI_PM_PATH	Path to the PubMed package
ORBACUS_LIBPATH	$ORBACUS_LIBPATH	Path to the ORBacus CORBA libraries
ORBACUS_INCLUDE	$ORBACUS_LIBPATH	Path to the ORBacus CORBA headers

Table 5. Compiler, Linker, and other development Tools

Macro	Source	Synopsis
CC	$CC	C compiler
CXX	$CXX	C++ compiler
LINK	$CXX	Linker (C++-aware)
CPP	$CPP	C preprocessor
CXXCPP	$CXXCPP	C++ preprocessor
AR	$AR	Library archiver
STRIP	$STRIP	Tool to strip symbolic info from binaries
RM	rm -f	Remove file(s)
RMDIR	rm -rf	Remove file(s) and directory(ies) recursively
COPY	cp -p	Copy file (preserving the modification time)

Macro	Source	Synopsis
CC_FILTER	@CC_FILTER@	Filters for the C compiler
CXX_FILTER	@CXX_FILTER@	Filters for the C++ compiler
CHECK_ARG	@CHECK_ARG@	
LN_S	@LN_S@	Make a symbolic link if possible; otherwise, hard-link or copy
BINCOPY	@BINCOPY@	Copy a library or an executable -- but only if it was changed

Example Makefiles

Below are links to examples of typical *makefiles*, complete with descriptions of their content.

- Inside the Tree
  - An example meta-makefile and its associated project makefiles
  - Library project makefile: Makefile.myProj.lib
  - Application project makefile: Makefile.myProj.app
  - Custom project makefile: Makefile.myProj
- New Projects and Outside the Tree
  - Use Shell Scripts to Create Makefiles
  - Customized makefile to build a library
  - Customized makefile to build an application
  - User-defined makefile to build... whatever